

JS Level 1



Введение



Модификация DOM

В этой лекции мы с вами будем разбирать "продвинутые" темы:

1. Генерацию элементов
2. Работу с формами
3. Удаление элементов
4. А также поговорим о некоторых аспектах JS

Поскольку наш курс подходит к концу, то нужно понимать, что сложность тем и сложность ДЗ будет возрастать (поскольку мы учимся делать всё более интересные вещи). Важный элемент вашего роста – это способность справляться со всё более сложными задачами и абстракциями.



Модификация DOM

Скорее всего, вы не "осилите" эту лекцию с одного раза и придётся работать с ней в несколько подходов.

Но вы ведь учитесь работать с большими объёмами информации, верно?



Повторение



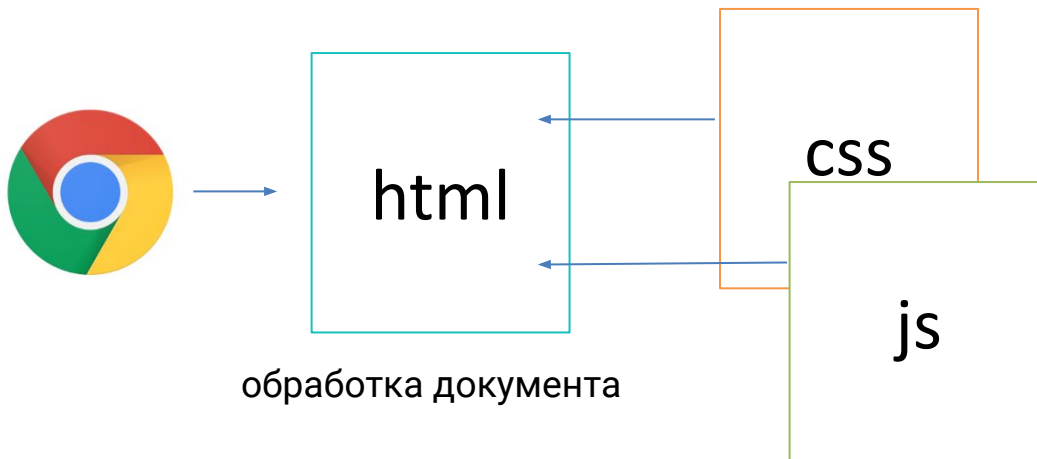
Web Application

На прошлой лекции мы поговорили с вами, как работают веб-приложения (а именно их клиентская часть) – они загружаются и запускаются в браузере:



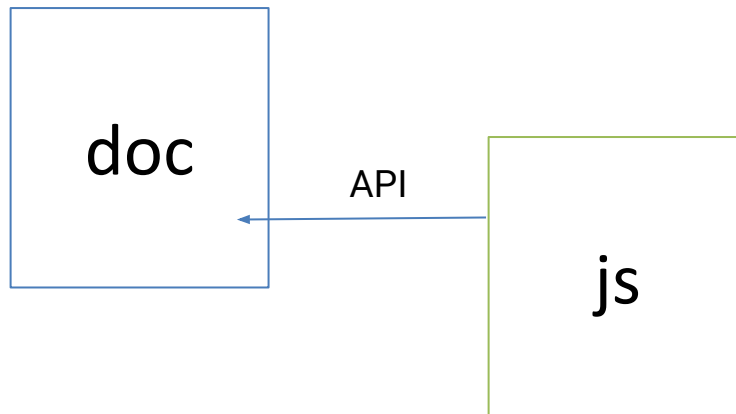
Ресурсы

Кроме того, мы обсудили сам механизм: сначала загружается HTML-документ (если вы указали его в адресной строке), а затем уже все ресурсы, которые в этом самом документе прописаны:



Ресурсы

Сегодня наша с вами задача – разобраться с тем, как мы можем прямо из JS создавать сам документ, а именно элементы внутри документа.



Задача



WishList

Мы хотим разместить в нашей социальной сети вот такой виджет, который позволяет пользователю писать о своих "желаниях" и, естественно, сразу указывать цену за каждое желание, а виджет оценивает сумму, которую нужно пользователю, чтобы исполнить все свои желания:

WishList

Название

Цена

Описание

Необходимо 2400 с.



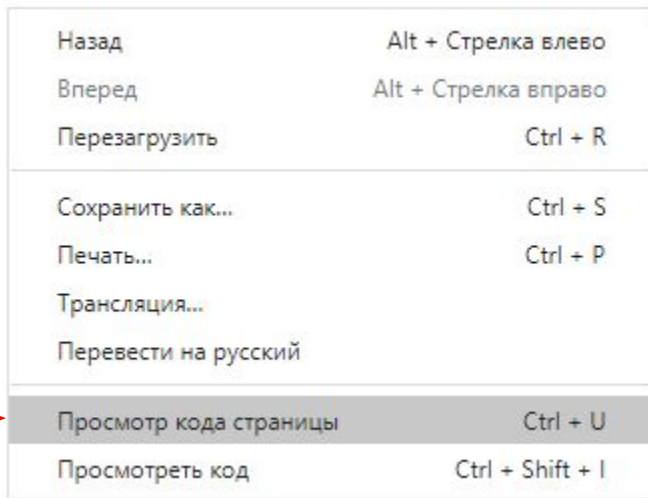
выглядит "страшненько" (но вы можете с помощью знаний Level 0 привести в нормальный вид)

- Название: Core i7, стоимость: 2400 с.



Код страницы

Начнём мы вот с такого трюка: зайдите на сайт <https://alif-skills.pro>, щёлкните правой кнопкой мыши и выберите "Просмотр кода страницы":



Это позволит вам увидеть, что присылает сервер браузеру в качестве документа. В то время как "Посмотреть код" показывает уже "обработанный браузером вариант".

Примечание: в Chrome к любому адресу достаточно добавить [view-source:](#) в адресной строке (например <view-source:https://alif-skills.pro>).



Загрузка страницы

Если убрать всё "лишнее", то мы получим примерно вот такую структуру:

```
<!doctype html>
<html lang="ru">

<head>
  <meta charset="utf-8" />
  <link rel="icon" href="/favicon.png" />
  <meta name="viewport" content="width=device-width,initial-scale=1" />
  <meta name="theme-color" content="#000000" />
  <meta name="description" content="Alif Skills" />
  <title>Alif Skills</title>
  <link href="/static/css/2.24bf1742.chunk.css" rel="stylesheet">
  <link href="/static/css/main.13e7c7d1.chunk.css" rel="stylesheet">
</head>

<body><noscript>Необходимо включить поддержку JavaScript в вашем браузере. Сервисы Alif Skills не работают без
  JavaScript.</noscript>
  <div id="root"></div>
  <script src="/static/js/2.fcb2f09a.chunk.js"></script>
  <script src="/static/js/main.ae2c8a65.chunk.js"></script>
</body>

</html>
```



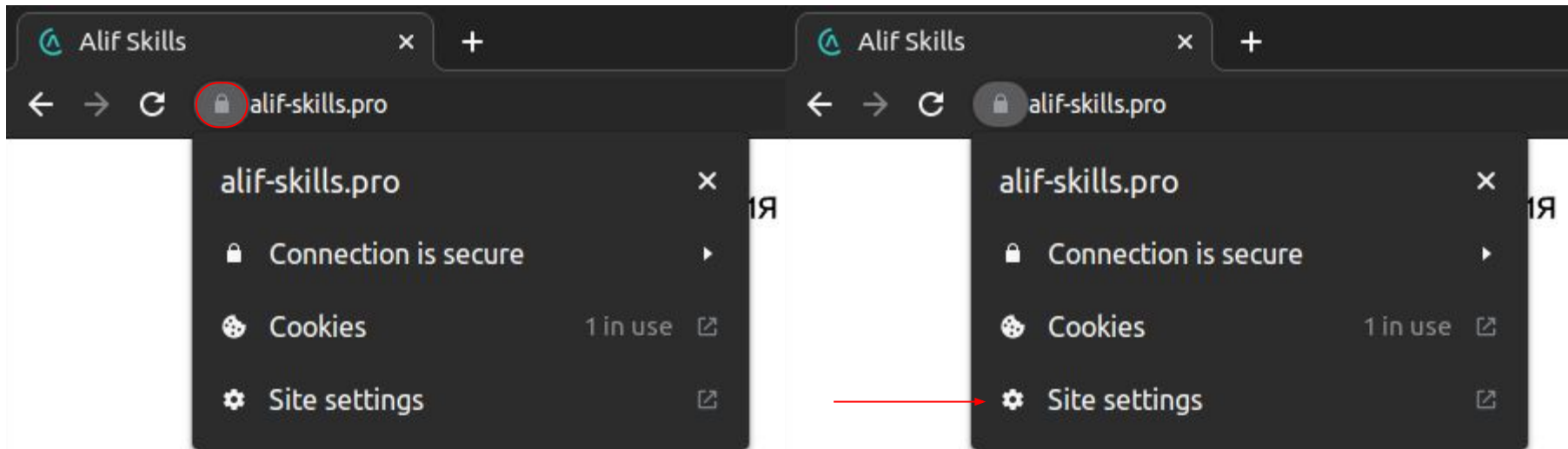
noscript

Элемент `noscript` отвечает за содержимого (контент), которое показывается в случае, если у пользователя выключен JS. Такое встречается достаточно редко и большинство сайтов не предусматривают работу в режиме отключенного JS, но оставить сообщение считается признаком хорошего тона (и в целом, профессионализма).



noscript

Как "попробовать выключить" JS? Для этого нужно кликнуть на указанную область в адресной строке браузера:











Настройки сайта

Вы попадёте на страничку настроек сайта, на которой можно выбрать, что разрешено, что запрещено, а что нужно спрашивать:

Разрешения

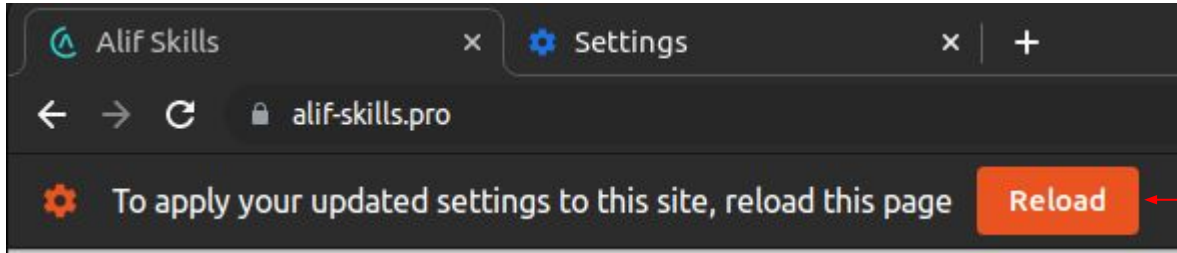
[Сбросить все разрешения](#)

 Геоданные	Спрашивать (по умолчанию ▼)
 Камера	Спрашивать (по умолчанию ▼)
 Микрофон	Спрашивать (по умолчанию ▼)
 Датчики движения	Разрешать (по умолчанию) ▼
 Уведомления	Спрашивать (по умолчанию ▼)
 JavaScript	Разрешать (по умолчанию) ▼
 Flash	<div>Разрешать (по умолчанию) Разрешить Блокировать</div>

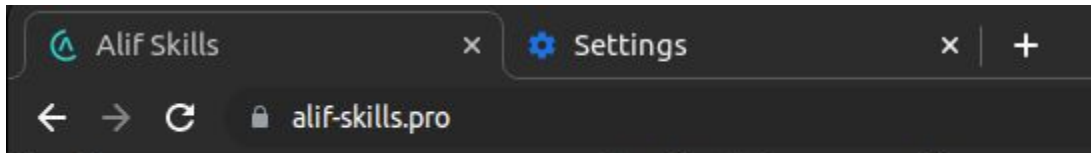


Применение настроек

После чего браузер попросит перезагрузить страницу:



И мы увидим "заветное" сообщение:



Необходимо включить поддержку JavaScript в вашем браузере.

Чтобы "вернуть всё как было", достаточно снова зайти на страницу настроек и нажать на "Сбросить все разрешения":



После чего снова перезагрузить страницу.



noscript

Итак, с **noscript** мы разобрались. Бот будет требовать от вас во всех задачах наличие этого тега с текстом о необходимости включения JS: "**Необходимо включить поддержку JavaScript в вашем браузере.**"



`<div id="root"></div>`

Остаётся лишь один вопрос: откуда берутся все элементы (например, форма входа), если внутри `body` остались только скрипты и `<div id="root"></div>`.

Если вы читали введение к лекции, то уже знаете, что они создаются средствами JS (с использованием DOM API, конечно же).



Создание элементов

DOM



Создание элементов

Итак, вспоминаем: самый главный наш объект внутри DOM – это `document`.

Именно он отвечает за текущий загруженный документ. И именно в этом объекте (или в цепочке его прототипов) содержится метод, позволяющий нам программно создавать элементы.

Называется он (этот метод) достаточно логично: `createElement`. Давайте посмотрим [на него в спецификацию](#) (напоминаем, в спецификации описаны интерфейсы, а не JS):

что возвращает название метода

параметры

```
Element createElement(  
  DOMString localName,  
  optional (DOMString or ElementCreationOptions) options = {}  
);
```



Создание элементов

В описании на MDN всё гораздо проще:

```
var element = document.createElement(tagName[, options]);
```

- `element` — созданный объект элемента.
- `tagName` — строка, указывающая элемент какого типа должен быть создан.
`nodeName` создается и инициализируется со значением `tagName`.
- `options` — необязательный параметр, объект `ElementCreationOptions`, который может содержать только поле `is`, указывающее имя пользовательского элемента, созданного с помощью `customElements.define()` (см. [Веб-компоненты](#)).

Т.е. мы вызываем этот метод, а в ответ нам возвращается созданный `Element`, с которым уже можно работать как с обычным элементом (как если бы он был создан из разметки).



Создание элементов

Что от нас хотят на самом деле? От нас хотят "название" элемента (опции мы с вами пока опустим).

Приступим, создадим следующую разметку:

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta name="viewport" content="width=device-width, initial-scale=1.0">
6      <title>Document</title>
7      <link rel="stylesheet" href="./css/styles.css">
8  </head>
9  <body>
10     <script src="./js/app.js"></script>
11 </body>
12 </html>
```



Создание элементов

Воспользуемся методом `document.createElement` и создадим элемент `h1`:

```
1  'use strict';  
2  
3  const headingEl = document.createElement('h1');
```

Если мы посмотрим на вкладку `Elements`, то увидим, ничего не увидим, почему?

```
<!doctype html>  
<html lang="en">  
  <head>...</head>  
  ...▼ <body> == $0  
    <script src="js/app.js"></script>  
    <!-- Code injected by live-server -->  
    <script type="text/javascript">...</script>  
  </body>  
</html>
```

Всё дело в том, что на странице отображаются только те элементы, которые находятся в дереве DOM.

Нашего элемента в этом дереве нет, т.к. дерево - это иерархия родитель-ребёнок, а мы только что созданный наш элемент никуда в этом дереве не определили (т.е. у него нет родителя). Нужно ему "найти" родителя.



Родители

Любая нода в DOM-дереве может быть родителем. У всех родителей есть несколько ключевых методов (надеюсь, вы помните, чем ноды отличаются от элементов):

```
[CEReactions] Node insertBefore(Node node, Node? child);  
[CEReactions] Node appendChild(Node node);  
[CEReactions] Node replaceChild(Node node, Node child);  
[CEReactions] Node removeChild(Node child);
```

По порядку:

1. `insertBefore` – можно вставить дочернюю ноду до другой
2. `appendChild` – можно добавить ноду в "конец списка детей"
3. `replaceChild` – можно заменить одного ребёнка другим
4. `removeChild` – можно удалить ребёнка (тогда он исчезнет со страницы)

Примечание: да, DOM API жестоко 😈, но не воспринимайте близко к сердцу – это всего лишь ноды.



Родители

Поступим самым простым способом: возьмём `document.body` и добавим ему ребёнка:

```
1  'use strict';
2
3  const headingEl = document.createElement('h1');
4
5  document.body.appendChild(headingEl);
```

Вроде как получилось, но не совсем хорошо (хотелось бы, чтобы `h1` был первым ребёнком, а не после `script`):

```
<!doctype html>
<html lang="en">
  <head>...</head>
... <body> == $0
  <script src="js/app.js"></script>
  <h1></h1>
  <!-- Code injected by live-server -->
  <script type="text/javascript">...</script>
</body>
</html>
```

сюда не смотрите, Live Server всегда вставляет свой код последним



root

Чтобы этого избежать (а также ещё ряда побочных последствий) используют немного другой подход: первым ребёнком внутри `body` при создании страницы указывают `<div id="root"></div>` а затем уже работают с ним, как мы с вами и видели, когда смотрели исходный код страницы:

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta name="viewport" content="width=device-width, initial-scale=1.0">
6      <title>Document</title>
7      <link rel="stylesheet" href="css/styles.css">
8  </head>
9  <body>
10     div#root
11     <script div#root Emmet Abbreviation
12 </body>
13 </html>
```

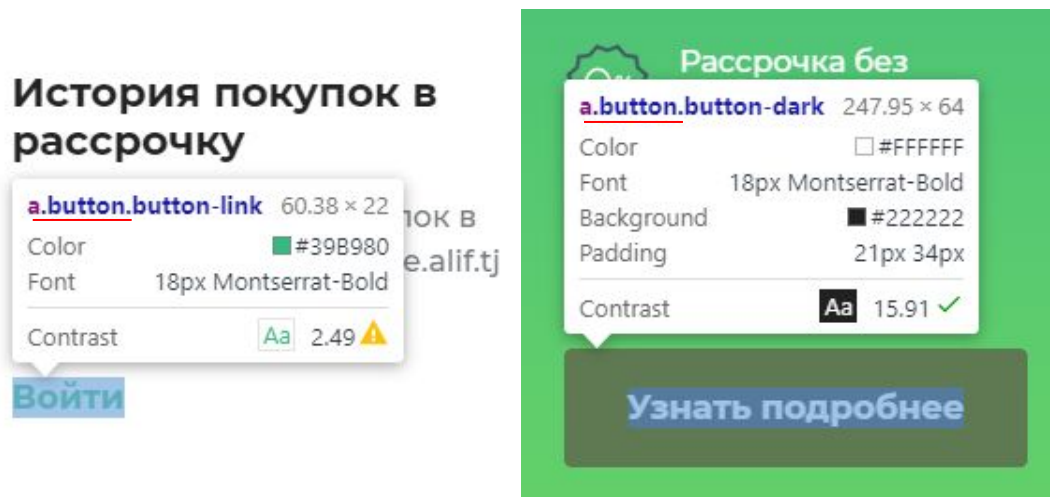
Примечание: `div#root` можно сократить до `#root` в Emmet.



Глобальные атрибуты

В рамках HTML существуют так называемые "глобальные" атрибуты – это атрибуты, которые можно применить к любому элементу. Нас будут интересовать только два:

1. **id** – атрибут, позволяющий назначить элементу на странице уникальный идентификатор
2. **class** – атрибут, содержащий (через пробел) список классов, используемых для отнесения элемента к какой-то общей группе (например, все "кнопки" на сайте <https://alif.tj> содержат класс **button**):

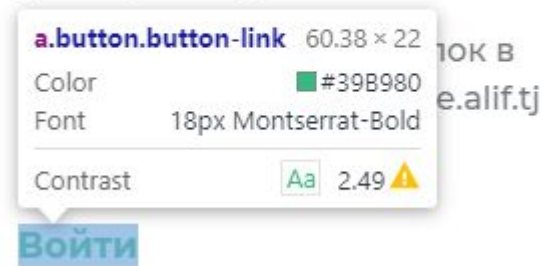


Глобальные атрибуты

Рассмотрим подробнее:

1. `a` – это элемент (ссылка)
2. `.button` – имя первого класса
3. `.button-link` – имя второго класса

История покупок в
рассрочку



Всё вместе в коде описано как:

```
<a href="https://online.alif.tj" class="button button-link">Войти</a>
```



Кнопка или ссылка?

У вас может возникнуть закономерный вопрос: "почему элементы гиперссылка, а класс у неё кнопка"?

Всё дело в том, что благодаря современным технологиям (в первую очередь CSS), любую кнопку можно оформить так, что вам будет казаться, что это гиперссылка, и наоборот.

И это касается не только кнопок и ссылок, а почти всех элементов (есть исключения, но о них не будем). Вам нужно запомнить следующее – задача Frontend'а сделать так, чтобы пользователю было удобно пользоваться вашим сервисом, а как вы это сделаете – это уже другой вопрос (вернитесь к лекциям Level 0 для детального рассмотрения этого вопроса). Поэтому многие вещи в вебе – не то, что кажется.



Глобальные атрибуты

Есть общие хорошие практики, которых следует придерживаться:

1. `id` используется для идентификации элемента (чтобы его можно было затем найти из скрипта) и для форм (об этом позже)
2. `class` нужно стараться использовать для стилизации из CSS

DOM API нам предоставляет методы для поиска по `id` и по классам:

- `document.getElementById('root');`
- `document.getElementsByClassName('button');`



Глобальные атрибуты

Аналоги через `querySelector`:

- `document.querySelector('#root');`
- `document.querySelector('.button');` – найдёт первый элемент с классом `button` в дереве
- `document.querySelectorAll('.button');` – найдёт все элементы с классом `button` в дереве



Emmet

Как вы уже могли заметить, сокращения Emmet иногда очень похожи на те выражения, которые мы используем в [querySelector](#). На самом деле всё потому, что разработчики Emmet отталкивались от них при создании.

Рекомендуем вам почитать [Emmet CheatSheet](#)* (шпаргалки), в которых вы ознакомитесь с ключевыми сокращениями.

Важно: сокращения Emmet хоть и похожи на селекторы, на самом деле не всегда ими являются! Будьте внимательны.



Создаём элементы

На самом деле, `getElementById` работает гораздо быстрее, чем `querySelector`.

Почему же мы не рассказывали вам о нём до этого? По двум причинам:

1. Эта скорость начинает играть роль, когда вы много раз выбираете элементы (а это плохо – мы говорили, что элементы нужно сохранять в константы)
2. Вам нужно было научиться пользоваться `querySelector`.

Но для разнообразия, в этот раз, будем использовать `getElementById`:

```
1  'use strict';
2
3  const rootEl = document.getElementById('root');
4  const headingEl = document.createElement('h1');
5  headingEl.textContent = 'WishList';
6  rootEl.appendChild(headingEl);
```

указали сразу, чтобы элемент не был пустым

```
<!doctype html>
<html lang="en">
  <head>...</head>
  ... <body> == $0
    <div id="root">
      <h1>WishList</h1>
    </div>
    <script src="js/app.js"></script>
    <!-- Code injected by live-server -->
    <script type="text/javascript">...</script>
  </body>
</html>
```

Удобство

А теперь представьте, что нам нужно сформировать вот такую структуру:

```
<div id="root">
  <form data-id="todo-form">
    <div>
      <label for="todo-text">Название</label>
      <input data-input="text" id="todo-text">
    </div>
    <div>
      <label for="todo-priority">Приоритет</label>
      <input data-input="priority" id="todo-priority" type="number">
    </div>
    <button data-action="add">Добавить</button>
  </form>
  <ul data-id="todo-list">
    <li data-todo-id="1">
      Выучить JS (приоритет: <span data-info="priority">1</span>)
      <button data-action="inc">+</button>
      <button data-action="dec">-</button>
    </li>
    <li data-todo-id="3">
      Выучить HTML (приоритет: <span data-info="priority">1</span>)
      <button data-action="inc">+</button>
      <button data-action="dec">-</button>
    </li>
    <li data-todo-id="2">
      Выучить CSS (приоритет: <span data-info="priority">2</span>)
      <button data-action="inc">+</button>
      <button data-action="dec">-</button>
    </li>
  </ul>
</div>
```

Попробуйте посчитать, сколько раз
придётся вызывать
`document.createElement` и `appendChild`.



Удобство

В реальных проектах (и фреймворках) конечно же пишут функцию, которая умеет из объекта делать такую структуру.

Когда же нужно "быстро" (не чтобы быстро работало, а чтобы быстро написать) иногда используют одну из ключевых возможностей браузера – превращать html-разметку в набор объектов.

Для этого у элементов есть волшебное свойство `innerHTML`, в которое можно записать "кусочек" html-разметки. Браузер распарсит эту разметку, превратит в элементы и сам подставит так, как будто бы мы сделали `appendChild`.

Способ этот, безусловно, удобный, но медленный и подвержен некоторым уязвимостям, но для полноты картины мы его покажем.



innerHTML

```
const rootEl = document.getElementById('root');  
const headingEl = document.createElement('h1');  
headingEl.textContent = 'WishList';  
rootEl.appendChild(headingEl);
```

vs

```
rootEl.innerHTML = `  
  <h1>WishList</h1>  
`;
```

Несколько ключевых моментов:

1. Мы использовали бэтики, поскольку именно они позволяют в JS записывать строки на "несколько строк"
2. На собеседованиях (в большинстве случаев) и в промышленном коде от вас будут ждать первый вариант, а не второй (а ещё лучше - функцию-шаблонизатор)

Примечание*: таким образом бэтики нужно использовать в двух случаях:

1. Когда вы подставляете в них значения через `${}` (template literals)
2. Когда пишете "многострочные строки"



Шаблонизатор

В одном из необязательных ДЗ мы попросим вас написать функцию-шаблонизатор, которая позволяет сложные объекты превращать в разметку (это одно из типовых заданий на собеседовании).

Пока же, мы сделаем всё "руками". Заголовок первого уровня (**h1**) мы создали, осталось разобраться с полями ввода, в которые пользователь будет вводить данные.



Формы



Формы

До этого мы с вами умели только кликать на элементах, сейчас же мы приступим к реализации получения данных от пользователя.

В нашем примере мы хотим, чтобы пользователь мог вводить название, описание и стоимость того, что он "желает".



Поля ввода

Для этого в используются поля ввода. На самом деле, ключевых элементов*, для того, чтобы организовать ввод данных от пользователя (помимо кнопок, конечно), три:

1. Поля ввода `input`
2. Многострочные поля ввода `textarea`
3. Списки выбора `select`

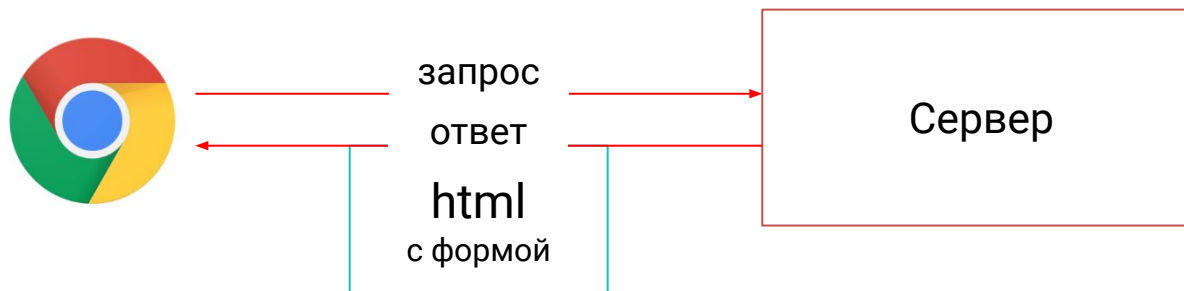
Остальное всё – либо подвиды `input`, либо некоторые продвинутые техники, которые мы разбирать пока не будем (например, атрибут `contenteditable`).



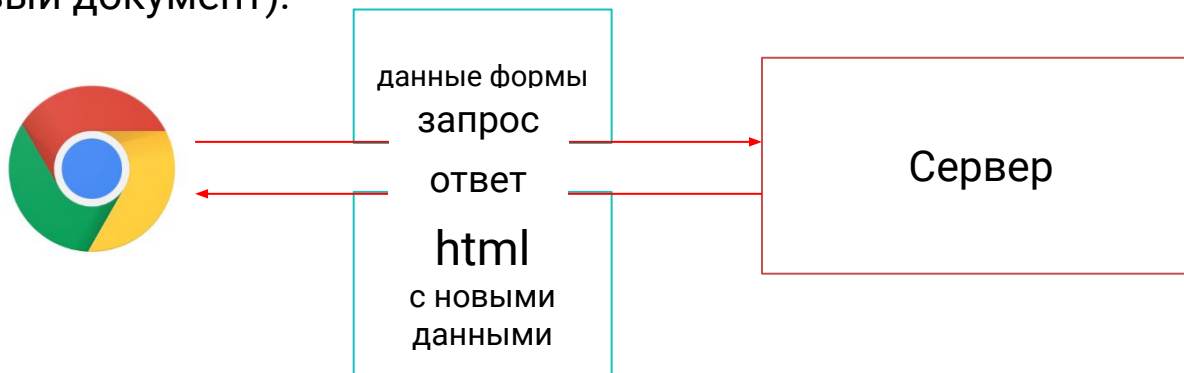
Формы

Изначально, когда JS мало использовали (у него было достаточно много проблем при работе в разных браузерах), общая схема выглядела следующим образом:

Шаг 1. Браузер отправляет запрос на сервер:



Шаг 2. Пользователь заполняет форму и отправляет на сервер (в ответ приходит новый документ):



Формы

Такая схема до сих пор широко распространена, но нас она пока особо не интересует.

Интересует лишь то, что для того, чтобы данные отправлялись на сервер, одним из условий было наличие элемента `form` – HTML-форма. Этот элемент (по умолчанию) никак визуально не отображается на странице и служит для объединения полей ввода в одну логическую единицу – форму.

Давайте создадим её:

```
1  'use strict';
2
3  const rootEl = document.getElementById('root');
4
5  const headingEl = document.createElement('h1');
6  headingEl.textContent = 'WishList';
7  rootEl.appendChild(headingEl);
8
9  const formEl = document.createElement('form');
10 rootEl.appendChild(formEl);
11
```

← не пренебрегайте пустыми строками
для логического разделения кода



input & textarea



input

4.10.5 The `input` element

4.10.5.1 States of the `type` attribute

- 4.10.5.1.1 Hidden state (`type=hidden`)
- 4.10.5.1.2 Text (`type=text`) state and Search state (`type=search`)
- 4.10.5.1.3 Telephone state (`type=tel`)
- 4.10.5.1.4 URL state (`type=url`)
- 4.10.5.1.5 E-mail state (`type=email`)
- 4.10.5.1.6 Password state (`type=password`)
- 4.10.5.1.7 Date state (`type=date`)
- 4.10.5.1.8 Month state (`type=month`)
- 4.10.5.1.9 Week state (`type=week`)
- 4.10.5.1.10 Time state (`type=time`)
- 4.10.5.1.11 Local Date and Time state (`type=datetime-local`)
- 4.10.5.1.12 Number state (`type=number`)
- 4.10.5.1.13 Range state (`type=range`)
- 4.10.5.1.14 Color state (`type=color`)
- 4.10.5.1.15 Checkbox state (`type=checkbox`)
- 4.10.5.1.16 Radio Button state (`type=radio`)
- 4.10.5.1.17 File Upload state (`type=file`)
- 4.10.5.1.18 Submit Button state (`type=submit`)
- 4.10.5.1.19 Image Button state (`type=image`)
- 4.10.5.1.20 Reset Button state (`type=reset`)
- 4.10.5.1.21 Button state (`type=button`)

`input` – это поля ввода. У этого замечательного элемента есть атрибут `type`, который позволяет этому элементу выглядеть совершенно по-разному, например:

`type="text"` – обычный текст:

`type="range"`:

`type="color"`:



input

Мы рекомендуем вам самостоятельно ознакомиться с типами [input](#) [на странице MDN](#). Обратим лишь ваше внимание на то, что не все типы хорошо поддерживаются браузером (а когда браузер не поддерживает какой-тип он отображает поле с типом [text](#)). Посмотреть общую поддержку конкретного типа вы можете на специальном сайте, который называется [caniuse](#). На этом сайте собирается информация о поддержке браузерами тех или иных возможностей.



input

Мы будем использовать:

- `input type="text"` для названия (обратите внимание, `text` итак по умолчанию, можно не писать, напоминаем, что вы можете узнать об этом из спецификации)
- `input type="number"` для стоимости (позволяет вводить только цифры)

Кроме того, нас будет интересовать ещё один элемент – `label`. Зачем? Сейчас увидите.



name

```
12  const nameContainerEl = document.createElement('div');
13  formEl.appendChild(nameContainerEl);
14
15  const nameLabelEl = document.createElement('label');
16  nameLabelEl.textContent = 'Название';
17  nameLabelEl.htmlFor = 'name-input';
18  nameContainerEl.appendChild(nameLabelEl);
19
20  const nameEl = document.createElement('input');
21  nameEl.id = 'name-input';
22  nameContainerEl.appendChild(nameEl);
```

▼ <form>

▼ <div>

<label for="name-input">Название</label>

<input id="name-input">

</div>

</form>

Основные моменты:

1. **label** – это специальный элемент-подпись для поля ввода (чтобы пользователь не забыл, что нужно ввести)
2. **label** "связывается" с конкретным полем ввода через атрибут **for** (он должен быть равен **id** поля ввода, но поскольку **for** в JS – это зарезервированное слово (для циклов), свойство в виде исключения называется **htmlFor**)
3. При клике на **label** курсор автоматически ставится в "привязанное" поле ввода



name

Проверим: кликните на тексте "Название" (1) - фокус переместится в поле ввода (2)

WishList

1 → Название

WishList

Название

2 ↑

Когда говорят, что фокус переместился на какое-то поле (ещё говорят фокус установлен на поле или поле в фокусе) это означает, что если сейчас пользователь будет что-то вводить с клавиатуры, то этот ввод будет направлен именно на этот элемент. В один момент времени только один элемент может быть в фокусе.



name

Фокус может устанавливаться не только на поле ввода, например, можно установить фокус на кнопке или гиперссылке. Тогда нажатие клавиши **Enter** приведёт к клику по кнопке или гиперссылке.



price

```
24  const priceContainerEl = document.createElement('div');
25  formEl.appendChild(priceContainerEl);
26
27  const priceLabelEl = document.createElement('label');
28  priceLabelEl.textContent = 'Цена';
29  priceLabelEl.htmlFor = 'price-input';
30  priceContainerEl.appendChild(priceLabelEl);
31
32  const priceEl = document.createElement('input');
33  priceEl.id = 'price-input';
34  priceEl.type = 'number'; ← указали type
35  priceContainerEl.appendChild(priceEl);
```



textarea

Для многострочных полей ввода (`input` позволяет вводить только одну строку) используется специальный элемент – `textarea`. Его использование в целом не отличается от `input`, разве что `type` писать не нужно + у него есть дополнительный атрибут `rows`, который указывает, сколько строчек для ввода отображать (пользователь может ввести и больше):

```
37  const descriptionContainerEl = document.createElement('div');
38  formEl.appendChild(descriptionContainerEl);
39
40  const descriptionLabelEl = document.createElement('label');
41  descriptionLabelEl.textContent = 'Описание';
42  descriptionLabelEl.htmlFor = 'description-input';
43  descriptionContainerEl.appendChild(descriptionLabelEl);
44
45  const descriptionEl = document.createElement('textarea');
46  descriptionEl.id = 'description-input';
47  descriptionEl.rows = 5;
48  descriptionContainerEl.appendChild(descriptionEl);
```



button

Остался последний штрих – добавить кнопку:

```
50  const addEl = document.createElement('button');  
51  addEl.textContent = 'Добавить';  
52  formEl.appendChild(addEl);
```



innerHTML

А теперь всё то же самое с `innerHTML` (он вас всегда будет соблазнять):

```
1  'use strict';
2
3  const rootEl = document.getElementById('root');
4  rootEl.innerHTML = `
5      <h1>WishList</h1>
6      <form>
7          <div>
8              <label for="name-input">Название</label>
9              <input id="name-input">
10         </div>
11         <div>
12             <label for="price-input">Цена</label>
13             <input id="price-input" type="number">
14         </div>
15         <div>
16             <label for="description-input">Описание</label>
17             <textarea id="description-input" rows="5"></textarea>
18         </div>
19         <button>Добавить</button>
20     </form>
21 `;
```

Но есть и минусы: у нас нет имён, которые бы ссылались на элементы внутри, а значит, придётся их (элементы) искать через `querySelector`.



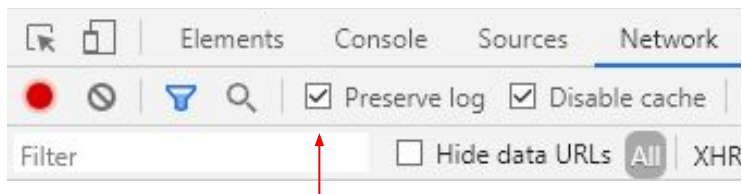
Обработка форм






Обработка формы

Итак, всё здорово, вы можете заполнить форму, нажать на кнопку "Добавить" и ... Форма очиститься. Что же произошло?

Откроем панельку **Network** и поставим флажок "**Preserve log**" (сохранять лог или журнал):



Попробуем снова заполнить форму (затем нужно нажать "Добавить") и увидим

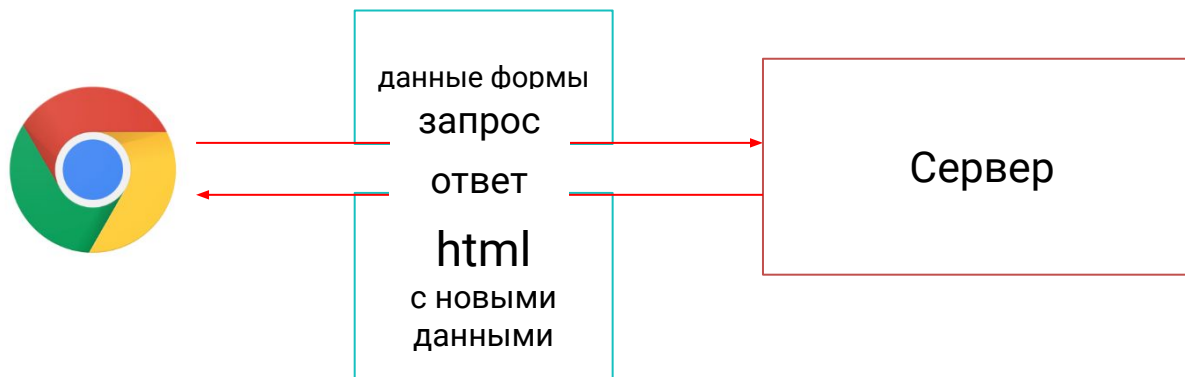
 index.html	200	docum...	Other	2.2 kB	7 ms
 styles.css	200	stylesh...	index.html	319 B	5 ms
 app.js	200	script	index.html	2.7 kB	8 ms

Т.е. браузер заново загрузил документ, а вместе с документом и стили, и скрипт. А значит, всё, что было – исчезло и браузер просто заново построил страницу (при этом все объекты, которые были до этого стёрлись из памяти).



Обработка формы

Это произошло по одной простой причине: браузер работает по-старинке, вы нажимаете на кнопку "Добавить", он отправляет данные на сервер (а сервер у нас LiveServer, который снова возвращает ту же страницу*):



Примечание*: на самом деле не всё так просто, но об этом мы поговорим на следующей лекции.



Событие

И тут мы начинаем вспоминать, что на прошлой лекции говорили с вами про события. И мы рекомендовали вам ознакомиться с событиями форм:

События формы

Имя события	Происходит когда
<code>reset</code>	Кнопка сброса нажата
<code>submit</code>	Кнопка "Отправить" нажата



Событие

Переходим на страничку описания события submit (к сожалению, она не переведена):

The `submit` event fires when a `<form>` is submitted.

Bubbles	Yes (although specified as a simple event that doesn't bubble)
Cancelable	Yes 1
Interface	<code>SubmitEvent</code>
Event handler property	<code>GlobalEventHandlers.onSubmit</code> 2



Событие

The `submit` event fires when a `<form>` is submitted.

Bubbles	Yes (although specified as a simple event that doesn't bubble)
Cancelable	Yes ¹
Interface	<code>SubmitEvent</code>
Event handler property	<code>GlobalEventHandlers.onSubmit</code> ²

Во-первых, нам говорят, можно отменять поведение этого события по умолчанию или нет (1). Напоминаем, что поведение по умолчанию – это то, что делает браузер если вы не вмешиваетесь (через JS).

Во-вторых, написано с помощью какого обработчика (2) можно это событие обрабатывать.

Ну и в-третьих, в тексте описания будет написано, что событие генерируется только на элементе `form`.



preventDefault

`preventDefault` – это специальный метод на объекте события, который просит браузер отменить поведение по умолчанию. Почему просим? Потому что есть события, для которых можно просить отменить поведение по умолчанию, но браузер не отменит (для таких в карточке будет написано Cancelable – No).

```
72  formEl.onSubmit = evt => {  
73    |    evt.preventDefault(); ←  
74    |    console.log(evt);  
75  |  };
```

Важно: не надо "лепить" `preventDefault` на каждое событие, т.к. это будет служить показателем вашей некомпетентности.

Если вы всё сделали аккуратно, ничего никуда отправляться не будет и страница не перезагрузится. А это именно то, чего мы хотели.



Объект события

Полное описание свойств и методов объекта события вы можете найти [на странице спецификации](#). Мы же будем рассматривать лишь ключевые свойства:

```
app.js:74
SubmitEvent {isTrusted: true, submitter: button,
▼ type: "submit", target: form, currentTarget: form
, ...} ⓘ
  bubbles: true
  cancelBubble: false
  cancelable: true
  composed: false
  currentTarget: null
  defaultPrevented: true
  eventPhase: 0
  isTrusted: true
▶ path: (6) [form, div#root, body, html, ...
  returnValue: false
▶ srcElement: form
▶ submitter: button
▶ target: form
  timeStamp: 391744.830000001055
  type: "submit"
▶ __proto__: SubmitEvent
```

Полное описание свойств и методов объекта события вы можете найти [на странице спецификации](#). Мы же будем рассматривать лишь ключевые свойства:

- **cancelable** – можно отменять поведение по умолчанию
- **defaultPrevented** – поведение по умолчанию отменено
- **type** – тип события



console.log

Если вы достаточно внимательный человек, то обратите внимание на одну



На самом деле это не странность, это особенность `console.log`. Знание этой особенности поможет вам сэкономить часы отладки. Всё дело в том, что верхнюю строку (в "свёрнутом" виде) `console.log` печатает тогда, когда событие происходит (т.е. когда вы вызываете `console.log`).

А то, что мы видим, когда кликаем на треугольник – это то, что хранится в объекте в тот момент, когда мы его "разворачиваем". Поэтому их "содержимое" может отличаться.



Извлечение значений

Всё это хорошо, но как получить доступ к тем значениям, которые ввёл пользователь? Здесь всё достаточно просто: первоначальные значения хранятся в атрибуте `value` самих элементов (а мы его не указывали – значит там "пустое" значение), а получить текущее значение (то, которое ввёл пользователь) можно с помощью свойства `value`:

```
72  formEl.onSubmit = evt => {  
73      evt.preventDefault();  
74  
75      const name = nameEl.value;  
76      const price = priceEl.value;  
77      const description = descriptionEl.value;  
78  
79      const wish = {  
80          name,  
81          price,  
82          description,  
83      };  
84  
85      console.log(wish);  
86  };
```

пока не генерируем id



Хранение

Осталось не так много:

1. Сохранить где-то полученный объект (возможно, массив подойдёт?)
2. Создать список
3. Сосчитать сумму



Хранение

Начнём с простого: создадим массив, в котором будем хранить все желания.

Нужно только определиться: новые мы будем добавлять в начало (метод `unshift`) или в конец (метод `push`) массива.

Давайте мы будем добавлять в начало:

```
72  const wishes = []; ←
73  formEl.onSubmit = evt => {
74      evt.preventDefault();
75
76      const name = nameEl.value;
77      const price = priceEl.value;
78      const description = descriptionEl.value;
79
80      const wish = {
81          name,
82          price,
83          description,
84      };
85      wishes.unshift(wish); ←
86  };
```



Сумма

Следующая по сложности задача – сосчитать и вывести сумму. Здесь у нас два варианта:

1. Хранить сумму в отдельной переменной и просто добавлять туда (либо вычитать)
2. Каждый раз пересчитывать

Как решать первую задачу вы уже знаете (вы делали это в рамках ДЗ с тем же звуком видеоплеера или магазином), поэтому мы посмотрим на второй вариант.

Самый простой вариант в данном случае, обойтись циклом (`for-of`), либо использовать `forEach` и замыкания. Но мы рассмотрим вариант поинтереснее.



filter, map, reduce



filter, map, reduce

У массива есть три замечательных метода (надеемся, что вы помните, что они хранятся в прототипе), о которых вас обязательно спросят на собеседовании (и они вам обязательно понадобятся в React):

1. **filter** – на базе существующего массива создаёт новый, в котором содержатся только те элементы, которые проходят по условию (т.е. количество элементов может поменяться, сами элементы меняться не должны*)
2. **map** – на базе существующего массива создаёт новый, в котором содержится столько же элементов*, но их тип может быть другим
3. **reduce** – сворачивает весь массив в одно результирующее значение (тип результирующего значения может быть какой угодно).

Примечание*: это JS, поэтому вы можете делать почти всё, что хотите, но это плохая практика – **filter** не должен менять элементы, **map** не должен менять количество.



filter

С **filter** вы уже знакомы, работает он примерно следующим образом. Мы посмотрим всё на детском примере: у нас есть шарики двух цветов (и на каждом шарике ещё написана цифра):



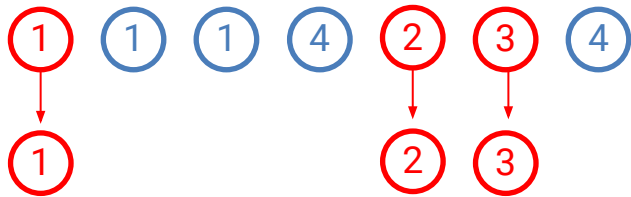
В JS это может быть представлено вот так:

```
const balloons = [  
  {num: 1, color: 'red'},  
  {num: 1, color: 'blue'},  
  {num: 1, color: 'blue'},  
  {num: 4, color: 'blue'},  
  {num: 2, color: 'red'},  
  {num: 3, color: 'red'},  
  {num: 4, color: 'blue'},  
];
```



filter

"Оставим" только красные шарiki (на самом деле - создадим новый массив):



Т.е. сами элементы не поменялись, но количество изменилось, т.к. те, кто не прошли по условию, в результирующий массив не попали.

В JS это может быть представлено вот так:

```
const redBalloons = balloons.filter(o => o.color == 'red');
```

↑
функция, по результату которой решается:
попадает элемент в итоговый массив или нет



map

`map` позволяет нам преобразовать массив, создав из каждого элемента какое-то новое представление. Например, мы хотим из массива шаров сделать массив чисел, которые написаны на этих шарах (обратите внимание - это снова будет новый массив, со старым ничего не случится):



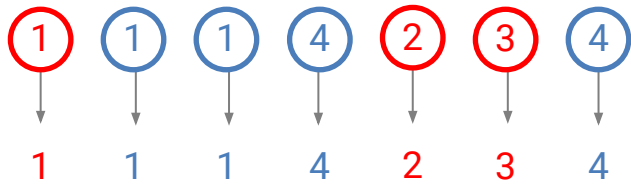
В JS это может быть представлено вот так:

```
const balloons = [  
  {num: 1, color: 'red'},  
  {num: 1, color: 'blue'},  
  {num: 1, color: 'blue'},  
  {num: 4, color: 'blue'},  
  {num: 2, color: 'red'},  
  {num: 3, color: 'red'},  
  {num: 4, color: 'blue'},  
];
```



map

Преобразуем шарiki в числа (которые были написаны на этих шариках):



Т.е. сами элементы поменялись, но количество не изменилось.

В JS это может быть представлено вот так:

```
const balloonsNumbers = balloons.map(o => o.num);
```

функция, по результату которой решается:
что именно попадает в результирующий массив



А как же дебаггер?

Это очень правильный вопрос, жаль вы его не задали, когда работали с `filter`.

Давайте посмотрим, как отлаживать подобные функции: начало стандартное – ставим точку остановки:

```
1 ▼ const balloons = [  
2   { num: 1, color: 'red' },  
3   { num: 1, color: 'blue' },  
4   { num: 1, color: 'blue' },  
5   { num: 4, color: 'blue' },  
6   { num: 2, color: 'red' },  
7   { num: 3, color: 'red' },  
8   { num: 4, color: 'blue' },  
9 ];  
10  
11 const redBalloons = balloons.filter(o => o.color === 'red');
```


кликаем сюда

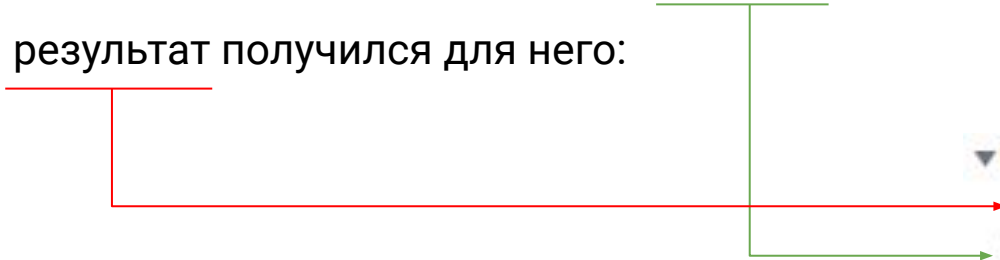


И тут у нас внутри вызова появляются дополнительные точки, которые нам могут быть интересны. Точки снаружи надо снять также кликом.



А как же дебаггер?

Теперь, обновив страницу и нажимая на кнопку  ,
мы будем видеть, какой элемент был на входе и
какой результат получился для него:



```
▼ Local  
Return value: true  
▶ o: {num: 1, color: "red"}  
this: undefined
```



reduce

С `reduce` обычно сложнее всего, т.к. его трудно понять с первого раза. Поэтому мы тут возьмём сначала даже не шарики, а просто массив чисел (тех, которые достались нам от `map`'а):

1 1 1 4 2 3 4

У `reduce` две формы:

1. С начальным значением
2. Без начального значения

Давайте смотреть:

```
const numbers = [1, 1, 1, 4, 2, 3, 4];
```

```
const sum = numbers.reduce((prev, curr) => prev + curr, 0);
```



reduce по шагам

На первом шаге в **prev** кладётся начальное значение – 0, а в **curr** первый элемент – 1, дальше (на следующую итерацию) проходит их сумма – 1

```
const numbers = [1, 1, 1, 4, 2, 3, 4];  
const sum = numbers.reduce((prev, curr) => prev + curr, 0);
```

▼ Local
Return value: 1
curr: 1
prev: 0

На втором шаге в **prev** кладётся результат предыдущего шага – 1, в **curr** второй элемент – 1, дальше проходит их сумма – 2

▼ Local
Return value: 2
curr: 1
prev: 1

И так далее (обязательно поэкспериментируйте в дебаггере). В итоге мы получим нашу сумму в виде одного числа.



reduce

Второй вариант **reduce** без начального значения – в этом случае произойдёт изменение только в первом шаге: в **prev** будет положен сразу первый элемент массива, а в **curr** – второй. Дальше всё так же.



reduce

Вам может показаться, что это всё чрезмерное усложнение, ведь можно было обойтись и циклом.

Ответ только один – привыкайте, это современный стиль работы (что в JS, что в других языках).



Возвращаемся к задаче

Итого, сосчитать итоговую сумму мы можем либо связкой `map` + `reduce`, либо `reduce` с начальным значением (подумайте, почему так):

```
72  const wishes = [];  
73  formEl.onSubmit = evt => {  
74      evt.preventDefault();  
75  
76      const name = nameEl.value;  
77      const price = priceEl.value;  
78      const description = descriptionEl.value;  
79  
80      const wish = {  
81          name,  
82          price,  
83          description,  
84      };  
85      wishes.unshift(wish);  
86  
87      const sum = wishes.map(o => o.price).reduce((prev, curr) => prev + curr);  
88  };
```

первый вариант



Возвращаемся к задаче

второй вариант

```
72  const wishes = [];  
73  formEl.onSubmit = evt => {  
74      evt.preventDefault();  
75  
76      const name = nameEl.value;  
77      const price = priceEl.value;  
78      const description = descriptionEl.value;  
79  
80      const wish = {  
81          name,  
82          price,  
83          description,  
84      };  
85      wishes.unshift(wish);  
86  
87      const sum = wishes.reduce((prev, curr) => prev + curr.price, 0);  
88  };
```



Вывод суммы

Осталось только вывести сумму:

```
72  const totalEl = document.createElement('div');
73  rootEl.appendChild(totalEl);
74
75  const wishes = [];
76  formEl.onSubmit = evt => {
77    evt.preventDefault();
78
79    const name = nameEl.value;
80    const price = priceEl.value;
81    const description = descriptionEl.value;
82
83    const wish = {
84      name,
85      price,
86      description,
87    };
88    wishes.unshift(wish);
89
90    const sum = wishes.reduce((prev, curr) => prev + curr.price, 0);
91    totalEl.textContent = `Необходимо ${sum} с.`;
92  };
```



Вывод суммы

И вот тут нас ждёт большой сюрприз: сумма считается неправильно:

WishList

Название	<input type="text" value="Core i7"/>
Цена	<input type="text" value="4200"/>
Описание	<input type="text" value="Новый процессор"/>
<input type="button" value="Добавить"/>	

Необходимо 04200 с.

Так ещё и форма не вычищается после добавления. Давайте разбираться по порядку.



input

Самое важное, что нужно запомнить: всё, что "приходит" из полей ввода – это текст (`string`). Даже если вы поставили тип `type="number"`, всё равно в `value` будет храниться текст и его нужно преобразовать к числу.

Вариантов для этого несколько:

1. "Хакерский": `+priceEl.value` – приведение строки к числу (поскольку `+` не бинарный, а унарный – работает только с одним операндом, то произойдёт преобразование к числу)
2. `parseInt(priceEl.value, 10)` или `Number.parseInt(priceEl.value, 10)` – специальная функция, пытающаяся разобрать строку и представить её в виде целого числа (`parseFloat` для вещественного)
3. `Number(priceEl.value)` – функция, используемая в том числе для преобразования значения в число (обратите внимание - функция называется именно `Number` и мы пишем её без `new`)



Унарный +

Тут всё достаточно просто, будет выглядеть вот так:

```
72  const totalEl = document.createElement('div');
73  rootEl.appendChild(totalEl);
74
75  const wishes = [];
76  formEl.onsubmit = evt => {
77    evt.preventDefault();
78
79    const name = nameEl.value;
80    const price = +priceEl.value;
81    const description = descriptionEl.value;
82
83    const wish = {
84      name,
85      price,
86      description,
87    };
88    wishes.unshift(wish);
89
90    const sum = wishes.reduce((prev, curr) => prev + curr.price, 0);
91    totalEl.textContent = `Необходимо ${sum} с.`;
92  };
```

В отличие от бинарного `+`, унарный всегда преобразует всё к числу и вы достаточно часто будете встречать такой подход.

Мы его не рекомендуем по одной простой причине: в JS сейчас много людей, пришедших из других языков, и для них – это не очевидно. А код надо писать так, чтобы всем в вашей команде было понятно.



Унарный +

Всё это работает благодаря тому, что в JS, как в математике, есть приоритет операторов. В нашем выражении `+priceEl.value` есть целых два оператора:

1. Унарный `+`
2. Доступ к свойствам `.`

Когда в выражении встречаются два оператора сначала выполняется тот, у которого приоритет выше, а затем тот, у кого ниже. Например в математике: $2 + 3 * 4$, у оператора умножения приоритет выше, поэтому сначала выполняется он и получаем $2 + 12 = 14$.

Здесь так же: у оператора `.` приоритет выше, поэтому если писать скобки, получилось бы вот так: `+(priceEl.value)`. Но они не нужны, если запомнить, что у `.` и `+` всегда максимальный приоритет.



Описание	Оператор
member	. []
call / create instance	() ^{вызов} new
negation/increment	! ~ _{-унарный} _{+унарный} ++ -- typeof void delete
multiply/divide	* / %
addition/subtraction	+ -
bitwise shift	<< >> >>>
relational	< <= > >= in instanceof
equality	== != === !==
bitwise-and	&
bitwise-xor	^
bitwise-or	
logical-and	&&
logical-or	
conditional	?:
assignment	= += -= *= /= %= <<= >>= >>>= &= ^= =
comma	,

max

приоритет

min



Number

Теперь давайте разбираться с `Number`. Это позволит нам создать почти полную "картинку" того, как всё устроено в JS. Начнём [со страницы MDN](#). Прочитав её, можно выделить несколько ключевых моментов:

1. `Number` – это объект
2. `Number` – это функция
3. Можно вызывать `Number` как `new Number(10)`, так и просто `Number(10)`
4. Есть методы и свойства вроде `Number.isNaN(10)` и `Number.NaN`
5. Есть методы и свойства вроде `Number.prototype.toFixed`
6. Ещё говорится про какие-то обёртки



Типы данных

Как вы уже знаете, в JS есть следующие типы данных:

1. Прimitives:

- a. **Boolean** – true/false
- b. **Null** – null (отсутствие значения*)
- c. **Undefined** – undefined
- d. **Number** – число
- e. **String** – строка
- f. **Symbol** – символы
- g. **BigInt** – большие целые числа

2. Объекты:

- a. **Object** – объект (функции – тоже объекты, но их можно ещё и вызывать с помощью оператора `()`)



Типы данных

Определить тип можно с помощью выражения `typeof value` или `typeof variable`:

```
> typeof 10
< "number"

> typeof 'Open School'
< "string"

> typeof {id: 10, name: 'Core i7'}
< "object"

> typeof document
< "object"

> typeof 10n
< "bigint"

> typeof undefined
< "undefined"
```



Типы данных

Это работает почти безотказно (всё, что не примитив – объект), но создатели языка решили, что для `null` `typeof` будет возвращать `'object'`, а для функций – `'function'`:

```
> typeof null
< "object"

> typeof console.log
< "function"

> typeof Number
< "function"
```

Соответственно, `Number` – это функция. А раз это функция, то её можно вызывать как функцию, например, `Number(priceEl.value)`:

```
const price = Number(priceEl.value);
```



Функция

Функция – это специальный тип объектов, которые можно "вызывать". Т.е. функция сама по себе является объектом, а значит у неё (у функции, могут быть свойства и методы). Просто так "распечатать" её не получится, т.к. в Chrome она написана на native code (т.е. не на JS, а например на C или C++), но доступ к свойствам и методам сохраняется:

```
> Number
< f Number() { [native code] }
> Number.MAX_VALUE
< 1.7976931348623157e+308
> Number.MIN_VALUE
< 5e-324
> Number.isNaN('Open School' * 2);
< true
```

Это надо для себя усвоить, что можно функцию вызывать (как на предыдущем слайде), а можно использовать "как объект" – т.е. хранить свойства и методы.



new

Чтобы понять идею с `new`, нам нужно отвлечься немного от чисел и вернуться, например, к "желаниям".

Мы с вами создаём желания следующим образом:

```
const wish = {  
  name,  
  price,  
  description,  
};
```

А теперь представьте, что мы делаем это не в одном участке кода, а, например, в 2 или 3. И в каком-то из них мы делаем вот так:

```
const wish = {  
  name,  
  price: 0,  
  description,  
};
```



new

```
const wish = {  
  name,  
  price,  
  description,  
};
```

```
const wish = {  
  name,  
  price: 0,  
  description,  
};
```

Так в чём проблема? Проблема в том, что мы легко можем опечататься в имени свойства (т.к. мы создаём эти объекты в разных местах) и тогда у нас получатся объекты с разным набором свойств:

```
const wish = {  
  name,  
  price,  
  description,  
};
```

прошло 50 строк кода

```
const wish = {  
  title,  
  price: 0,  
  description,  
};
```

С точки зрения JS – всё ок.



Функции

Мы вам уже говорили, что если какое-то действие повторяется больше одного раза, то лучше завернуть его в функцию, чтобы "не опечатываться". Так и здесь: почему бы нам просто не создать функцию, которая создаёт объекты нужного

```
function createWish(name, price, description) {  
  const object = {};  
  object.name = name;  
  object.price = price;  
  object.description = description;  
  return object;  
}
```

Мы специально записали всё именно в таком виде, чтобы вы поняли общую нить рассуждений:

1. Создаём новый пустой объект
2. В него добавляем свойства
3. Возвращаем объект



Функции

Если посмотреть на эту функцию, то становится понятным её предназначение – конструировать объект нужной формы. Поэтому решили сделать следующее: упростить нам запись, исключив дублирование кода:

```
function createWish(name, price, description) {  
  const object = {}; ←  
  object.name = name;  
  object.price = price;  
  object.description = description;  
  return object; ←  
}
```

Обратите внимание, какой бы мы объект не создавали (например, не желание, а пост), строки, отмеченные стрелками всегда будут одни и те же.



Функции

Поэтому решили ввести одно правило и одно соглашение.

Начнём с соглашения – оно заключается в том, что функции-конструкторы принято называть с большой буквы. А правило заключается в том, что если не просто вызывать функцию, а ставить перед ней ключевое слово **new**, то за нас создадут пустой объект и подставят **return**:

```
function createWish(name, price, description) {  
  const object = {};  
  object.name = name;  
  object.price = price;  
  object.description = description;  
  return object;  
}
```



```
function Wish(name, price, description) {  
  this.name = name;  
  this.price = price;  
  this.description = description;  
}
```

```
const wish = new Wish('Core i7', 2400, 'Мощный процессор');
```



Здесь важно отметить следующий момент: когда функция вызывается с **new**, в **this** кладётся **пустой новый объект**. Воспринимайте в данном случае **this** как удобное имя, которым решили назвать этот самый объект.



Объекты

Почему мы выделили слова **пустой новый объект**? До этого мы с вами об этом не разговаривали, но пришла пора это обсудить: объекты и примитивы живут по-разному. Например:

```
> 10 == 10  
< true  
  
> {} == {}  
< false
```

Но почему два числовых литерала равны друг другу, а два объекта – нет? Просто потому, что есть фундаментальная разница между примитивами и объектами.

Примитив – это просто значение. Чем 10 отличается от 10? Ничем. А объекты – это уникальные сущности. Например, возьмите два "одинаковых" автомобиля с завода – они действительно одинаковые. Но это разные объекты. Вы можете сравнивать их свойства и говорить, что свойства одинаковы, но это всё равно будут разные объекты.



Объекты vs Примитивы

Так вот оператор `==` для примитивов проверяет, что совпадают значения, а для объектов – что имена указывают на один и тот же объект. Что это значит? Давайте смотреть:

```
> let primitiveOriginal = 20;
< undefined

> let primitiveCopy = primitiveOriginal;
< undefined
```

← В `primitiveCopy` значение 20

```
> primitiveOriginal += 10;
< 30
```

← В `primitiveCopy` значение не изменилось

```
> primitiveCopy;
< 20
```



Объекты vs Примитивы

Когда же мы говорим об объектах, воспринимайте это как людей, вы можете давать одному и тому же объекту (оператор `=`) разные имена, но человек по-прежнему останется один:

```
> let objectOriginal = {name: 'Vasya'};
< undefined

> let objectCopy = objectOriginal;
< undefined

> objectOriginal.name = 'Vasiliy Petrovich';
< "Vasiliy Petrovich"

> objectCopy
< ▶ {name: "Vasiliy Petrovich"}
```



Объекты vs Примитивы

И чтобы закрепить, представляйте себе следующую аналогию:

- Примитивы – это неизменяемые вещи, когда вы пытаетесь записать в переменную примитив, он просто туда записывается(`originalValue += 10` это не попытка поменять 20, 20 нельзя изменить, 20 это число, это запись в `originalValue` нового числа 30)
- Объекты – это изменяемые вещи, не существует двух одинаковых объектов, существуют либо разные имена для одного и того же объекта (родители вас называют по-одному, а друзья – по-другому), либо объекты, у которых одинаковые все свойства или некоторые (но объекты эти – физически разные, вспомните аналогию с автомобилями)



Объекты vs Примитивы

Эту разницу важно усвоить, потому что она проявляется во всём:

```
> function changePrimitive(value) {  
    value++;  
}
```

```
< undefined
```

```
> let primitive = 10;
```

```
< undefined
```

```
> changePrimitive(primitive);
```

```
< undefined
```

```
> primitive;
```

```
< 10
```

Вот это value всё равно, что: `let value = primitive`.

Представьте, что вы дали номер доставки пиццы своему товарищу, а он взял и "поменял" в этом номере пару цифр. Что будет с тем номером, который записан у вас? Ничего, на самом деле, он просто записал себе "другой" номер.



Объекты vs Примитивы

Эту разницу важно усвоить, потому что она проявляется во всём:

```
> function changeObject(value) {  
    value.name = 'Vasiliy Petrovich';  
}  
< undefined  
> let object = {name: 'Vasya'};  
< undefined  
> changeObject(object);  
< undefined  
> object  
< ▶ {name: "Vasiliy Petrovich"}
```

Вот это `value` всё равно, что: `let value = object`, но объект всего один и мы меняем именно его свойства (объекты менять можно – числа, строки, `boolean` и другие примитивы – нет). Представьте, что вы отдали машину на ремонт. Если её там случайно поцарапают, то царапина никуда магическим образом не исчезнет – это ваш автомобиль и он один (нельзя для него сделать `copy & paste`).

Обратите внимание, что мы меняли именно свойство объекта. Если бы мы написали внутри функции `value = {}`, то с оригинальным объектом бы ничего не случилось (мы просто бы потеряли на него ссылку).



Number

В случае с **Number** это позволяет увидеть следующий набор примеров:

```
> Number('10') == Number('10') ← примитивы
< true

> new Number('10') == new Number('10') ← объекты
< false

> const obj = new Number('10');
< undefined

> obj
< ▼ Number {10} ⓘ
  ► __proto__: Number
  [[PrimitiveValue]]: 10
```

Таким образом, если мы хотим преобразовать строку в число, нам нужна форма без **new**.

Важно: следуйте соглашениям, если вы не предполагаете использование функции в качестве функции конструктора – не пишите её с большой буквы!



Number.prototype

С `prototype` всё просто – у функции есть свойство `prototype`. В ней хранится объект, который при вызове `new` устанавливается созданному объекту в свойство

`__proto__`. На примере наших желаний:

```
function Wish(name, price, description) {  
  this.name = name;  
  this.price = price;  
  this.description = description;  
  // this.__proto__ = Wish.prototype; - это делается за нас ←  
}
```

```
const wish = new Wish('Core i7', 2400, 'Мощный процессор');
```

С `Number` та же самая история:

```
> const obj = new Number('10');  
< undefined  
> obj.__proto__ == Number.prototype  
< true
```

Другой вопрос, зачем нам объект `Number`, когда есть примитивы? Об этом буквально через пару слайдов.



Прототипы

Делается это только для выстраивания цепочки прототипов (вспоминаем задачу с Таносом):



Array

На этом этапе вы должны задать вопрос: мы же никогда не вызывали `new Array`? На самом деле, форма `[1, 2, 3]` "аналогична" вызову `new Array(1, 2, 3)`. И, да, `Array` – это тоже функция-конструктор:

```
> const data = new Array(1, 2, 3);  
< undefined  
  
> data  
< ▶ (3) [1, 2, 3]  
  
> data.__proto__ == Array.prototype  
< true
```

автоматически будет выполнено:

`data.__proto__ = Array.prototype;`

И в свойстве `prototype` этой функции (вспоминаем, что функция - это объект) хранится объект, который записывается создаваемому массиву в `__proto__`.



Обёртки (wrappers)

Итак, в деле с `Number` остался последний вопрос "что такое обёртки"? Теперь мы с вами чётко знаем, что есть объекты и примитивы. У объектов есть свойства и методы (на самом деле методы – это просто функции, которые записаны в свойства). Например, метод `toFixed` позволяет получить строку из числа с нужным количеством цифр после запятой.



Обёртки (wrappers)

Методы умеют делать различную полезную работу и иногда хочется использовать методы с примитивами. Но вот так – это очень долго (1) поэтому дали возможность "опустить" часть кода и просто сделать в сокращённом виде (2), но "за сценой" всё происходит как слева:

①

```
> const obj = new Number(10.33333);  
< undefined  
  
> const result = obj.toFixed(2);  
< undefined  
  
> result;  
< "10.33"  
  
> typeof result;  
< "string"
```

②

```
> const result = (10.33333).toFixed(2);  
< undefined  
  
> result  
< "10.33"
```

↑
примитив "автоматически" завернули
в объект и на объекте вызвали метод
(круглые скобки нужны только для читабельности)



ОСНОВЫ

По факту, мы именно сейчас на базе всех знаний, которые у нас были до этого, разобрались как же всё устроено. Не поленитесь и перечитайте этот раздел несколько раз, пока у вас не сложится точного понимания.

Если же останутся вопросы – пишите в канал курса.



Валидация



Валидация

Задача ввода данных тесно связана с задачей валидации: мы хотим помочь пользователю замечать ошибки и исправлять их быстро (напоминаю, что пользователь может обойти всю защиту фронтенда и записать что хочет и куда хочет).

Для этого мы должны последовательно проверять все поля, которые он вводит, и, если что-то не так, сообщать ему об этом.

Примечание*: на самом деле, примерно 70% (а то и больше) вашей работы будет связано с валидацией, поэтому будем учиться.



Валидация

Есть специальное API, которое позволяет валидировать формы (включая возможности HTML), но мы пока обойдёмся чистым JS. Давайте думать, что может пойти не так*:

1. Пользователь может ввести в поле название пустую строку, или строку с пробелами (неплохо бы их убрать)
2. Пользователь может ввести отрицательное число или не число
3. Пользователь может не ввести описание или заполнить его пробелами

На самом деле, правила валидации целиком зависят от того, какое приложение вы пишете. Например, если вы просите ввести номер карты, то там есть специальные алгоритмы (см. алгоритм Луна), которые позволяют проверить – это номер карты или просто набор цифр.

Примечание*: ваши пользователи всегда будут удивлять вас своей



string

Итак, имя – это строка. И нам нужны удобные функции для удаления пробелов из строки. Совсем недавно мы с вами обсуждали обёртки (wrappers) для чисел и будет разумно предположить, что такие же есть и [для строк](#). Среди методов есть `trim` – позволяет удалять начальные и конечные пробельные символы:

```
> '  Core i7  '.trim()
< "Core i7"
```

И логика очень простая:

```
const wishes = [];
formEl.onSubmit = evt => {
  evt.preventDefault();

  let error = null;
  const name = nameEl.value.trim();
  if (name === '') {
    error = 'Заполните поле Название';
    console.log(error);
    // TODO: выделить для отображения error элемент
    nameEl.focus();
    return;
  }
}
```

устанавливаем фокус в поле с ошибкой
и выходим из функции



===

Q: что это за ===? Ведь всегда было ==?

A: ==, которое мы с вами до этого использовали, обладает одной нехорошей особенностью – оно приводит типы. А мы с вами говорим, что полагаться на встроенное поведение типов – нехорошо и мы сами приводим типы (вспомните Number), то теперь мы будем использовать ===, поскольку оно не приводит типы:

```
> '0' == 0;  
< true  
-----  
> '0' === 0;  
< false
```

Для проверки на неравенство у нас есть !== (не равно без приведения типов).

Начиная с сегодняшнего дня, использование == или != будет считаться ошибкой. После ДЗ вы найдёте раздел с общей информацией о приведении типов.



error

Для отображения ошибки (сообщения) нам нужен элемент. Тут есть два варианта:

1. Один общий элемент
2. На каждое поле ввода свой элемент

Конечно же, второй вариант лучше (и вы его отработаете в рамках ДЗ), мы же пойдём по первому варианту:

```
75  const errorEl = document.createElement('div');
76  formEl.insertBefore(errorEl, formEl.firstChild);
77
78  const wishes = [];
79  formEl.onsubmit = evt => {
80    evt.preventDefault();
81
82    errorEl.textContent = '';
83    let error = null;
84    const name = nameEl.value.trim();
85    if (name === '') {
86      error = 'Заполните поле Название';
87      errorEl.textContent = error;
88      nameEl.focus();
89      return;
90    }
```

вставляем до первого ребёнка
(т.е. теперь `errorEl` – первый ребёнок)
могли обойтись и `appendChild`,
если бы делали в самом начале

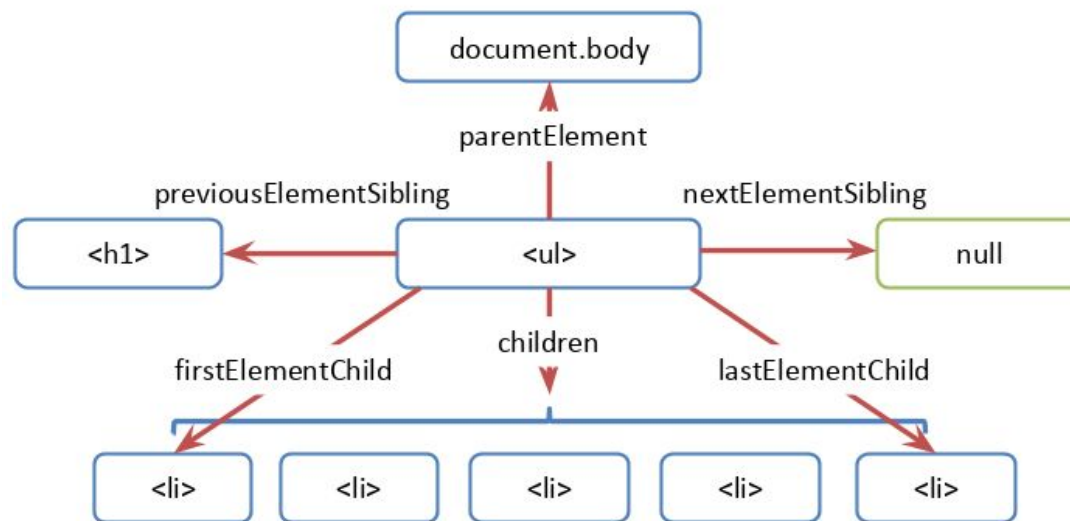
"очищаем" результаты предыдущей валидации



firstElementChild

Откуда взялся `firstElementChild`? Ведь мы о нём ничего не говорили? Почему мы просто не взяли `children[0]`? Полезно знать ключевые свойства DOM-элементов, упрощающие навигацию. Посмотрим на следующем примере:

```
10 <body>
11   <h1>Alif skills</h1>
12   <ul>
13     <li>1 item</li>
14     <li>2 item</li>
15     <li>3 item</li>
16     <li>4 item</li>
17   </ul>
18 </body>
```



относительно `ul`

Напоминаем, что вы всегда можете воспользоваться дебаггером и консолью, чтобы посмотреть "кем" один из элементов, приходится другому.



price

Вернёмся к валидации и разберёмся с `price`:

```
93     const price = Number(priceEl.value);
94     if (Number.isNaN(price)) {
95         error = 'Неверно введена цена';
96         errorEl.textContent = error;
97         priceEl.focus();
98         return;
99     }
100
101     if (price < 0) {
102         error = 'Цена не может быть отрицательной';
103         errorEl.textContent = error;
104         priceEl.focus();
105         return;
106     }
```

Мы для простоты убрали некоторые проверки, например, что пользователь ввёл очень большое число. Но в целом, всё выглядит вот так.



isNaN vs Number.isNaN

Если почитать документацию на MDN, то выяснится, что есть `isNaN` и `Number.isNaN`. Какой же из них использовать? Рекомендуется использовать именно `Number.isNaN`. В JS в последние годы начали наводить порядок и перемещать либо создавать аналоги функций, расположенных в глобальном объекте, в специализированные объекты. Вот так появился `Number.isNaN` (это не копия, это другая версия):

```
> isNaN === Number.isNaN  
< false
```



price

Вернёмся к валидации и разберёмся с `description` (ничего интересного нет, всё то же самое, что и с `name`):

```
108     const description = descriptionEl.value.trim();
109     if (description === '') {
110         error = 'Заполните поле Описание';
111         errorEl.textContent = error;
112         descriptionEl.focus();
113         return;
114     }
```



ИТОГИ

Неплохо, дело за малым – очистить форму после добавления (если все проверки прошли) и отображать конкретный элемент в списке покупок. А потом добавить ещё и удаление.



Очистка формы

Важно: очищать форму нужно только после успешного добавления. В противном случае пользователю придётся заново заполнять всю форму.

Самое большое, что не любят делать пользователи – это заполнять формы, поэтому максимально облегчайте им этот процесс.

Для очистки формы у нас есть метод `reset` (нужно поискать на MDN у элемента формы), который просто сбрасывает состояние формы до первоначального:

```
116     const wish = {  
117         name,  
118         price,  
119         description,  
120     };  
121     wishes.unshift(wish);  
122  
123     formEl.reset();
```



Список



Список

Для отображения списка, нам, как ни странно, нужен список. Список создаётся с помощью элемента `ul`, а элементами списка могут являться элементы `li`.

Создадим предварительно сам список и добавим его в `rootEl`:

```
78  const listEl = document.createElement('ul');
79  rootEl.appendChild(listEl);
```

А затем, при каждом клике будем добавлять элемент:

```
128  |   const rowEl = document.createElement('li');
129  |   rowEl.textContent = `Название: ${wish.name}, стоимость: ${wish.price} с.`;
130  |   listEl.insertBefore(rowEl, listEl.firstChild);
```



Список

Получилось неплохо (если не смотреть на оформление):

WishList

Название	<input type="text"/>
Цена	<input type="text"/>
Описание	<input type="text"/>
<input type="button" value="Добавить"/>	

Необходимо 2400 с.

Но наша функция разрослась (больше 50 строк кода) – это плохо. И что с этим делать, пока непонятно.

Мы научимся "разруливать" такие ситуации уже на следующем курсе, когда будем обсуждать классы, веб-компоненты и "всё такое".

- Название: Core i7, стоимость: 2400 с. Пока же вернёмся к удалению.



Удаление



Удаление

Для того, чтобы пользователь имел возможность что-то удалить, ему нужно дать какой-то управляющий элемент, например, кнопку:

```
128     const rowEl = document.createElement('li');
129     rowEl.textContent = `Название: ${wish.name}, стоимость: ${wish.price} с.`;
130     listEl.insertBefore(rowEl, listEl.firstChild);
131
132     const removeEl = document.createElement('button');
133     removeEl.textContent = 'Удалить';
134     removeEl.onclick = () => {
135
136     };
137     rowEl.appendChild(removeEl);
```

И теперь самый интересный вопрос: а как удалить? Он разбивается на два:

1. Как удалить элемент из DOM?
2. Как удалить элемент из массива?



Удаление из DOM

Для удаления из DOM есть два метода:

- `remove` – вызывается на той ноде, которую хотим удалить
- `removeChild` – вызывается на родителе, в качестве аргумента нужно передать ребёнка, которого нужно удалить

Вы спросите "зачем нужен второй"? Всё дело в том, что первый начал поддерживаться большинством браузеров "не так давно":

Chrome	Edge *	Safari	Firefox	Opera	IE	Chrome for Android	Safari on iOS *	Samsung Internet	Opera Mini *	Opera Mobile *	UC Browser for Android	Android Browser *	Firefox for Android	QQ Browser	Baidu Browser	KaiOS Browser
		3.1-5.1														
4-23	12	6	2-22	10-12.1			3.2-6.1					2.1-4.3				
24-109	13-109	6.1-16.2	23-109	15-94	6-10		7-16.2	4-18.0		12-12.1		4.4-4.4.4				
110	110	16.3	110	95	11	109	16.3	19.0	all	73	13.4	109	109	13.1	13.18	2.5
111-113		16.4-TP	111-112				16.4									

Поэтому на собеседованиях могут удивиться, если увидят его, а не `removeChild`.

Давайте посмотрим на `removeChild`.



Closures

Самое важное, что у нас есть – это замыкания: внутри обработчика клика на кнопке мы знаем и родителя и ребёнка (которого нужно удалить):

```
const removeEl = document.createElement('button');
removeEl.textContent = 'Удалить';
removeEl.onclick = () => {
  listEl.removeChild(rowEl);
};
rowEl.appendChild(removeEl);
```

эта строка срабатывает, когда пользователь
кликает на Удалить, а не когда элемент добавляется



Closures

Почему бы с массивом не поступить так же? Конечно можно попробовать, вот только у массива нет метода `removeChild`.

Как же быть тогда? Варианта два:

1. Есть метод `splice` (очень хитрый), который умеет по индексу удалять элемент (для этого ещё нужно найти индекс)
2. Есть ваш любимый метод `filter` (мы можем не менять массив, а просто создать новый, но тогда `wishes` должен быть объявлен через `let`, а не `const`)

Рассмотрим оба варианта.



splice

```
const removeEl = document.createElement('button');
removeEl.textContent = 'Удалить';
removeEl.onclick = () => {
  listEl.removeChild(rowEl);
  const index = wishes.indexOf(wish);
  wishes.splice(index, 1);

  // не забываем пересчитывать сумму
  const sum = wishes.reduce((prev, curr) => prev + curr.price, 0);
  totalEl.textContent = `Необходимо ${sum} с.`;
};
rowEl.appendChild(removeEl);
```



filter

```
const removeEl = document.createElement('button');
removeEl.textContent = 'Удалить';
removeEl.onclick = () => {
  listEl.removeChild(rowEl);
  wishes = wishes.filter(o => o !== wish);

  // не забываем пересчитывать сумму
  const sum = wishes.reduce((prev, curr) => prev + curr.price, 0);
  totalEl.textContent = `Необходимо ${sum} с.`;
};
rowEl.appendChild(removeEl);
```

Нам больше нравится вариант с **filter** (молодёжно, стильно, современно).



Итоги



ИТОГИ

В этой лекции мы обсудили самую важную тему, которая позволяет вам организовать взаимодействие с пользователями.

Мы обсудили только малую её часть, рассказав вам про Event Handler'ы. Помимо Event Handler'ов есть ещё Event Listener'ы – это возможность добавлять более одного обработчика одного и того же события на одном элементе. Кроме того, остаётся тема того, как событие "путешествует" по документу.

Этому всему будет посвящена наша следующая лекция.



Итоги

Финальный момент: поскольку мы делаем приложение на русском языке, то можно (и нужно) в `lang` прописать `"ru"`:

```
<!DOCTYPE html>  
<html lang="ru">
```



Домашнее задание



Орг.моменты

Практикум состоит из 8 обязательных занятий. Мы выкладываем новые занятия каждый понедельник в 14:00 (по Душанбе), кроме первой недели.

Каждое воскресенье в 23:59 (по Душанбе) дедлайн сдачи домашнего задания. Дедлайн – это предельный срок, до которого вы должны сдать ДЗ.

Если не успеете сдать в срок домашнее задание, тогда этот практикум будет для вас закончен и вы сможете зарегистрироваться на запуск следующего через несколько месяцев.

Все вопросы вы сможете задавать в [Телеграм канале](#).



Общие требования

Важно: для всех задач предполагается, что у вас на внутри `body` есть только `noscript` и `div` с `id="root"` (и, конечно же, подключение `app.js`). Всё остальное генерируется программно. Бот будет за этим следить.

После каждого добавления элемента форма должна вычищаться, а фокус устанавливаться в первое поле.

Ни одна форма не должна разрешать добавлять элементы с пустыми значениями: создайте элемент с `data-id="message"` и выводите туда сообщение: "Значение поля не может быть пустым" и устанавливайте на поле с ошибкой фокус (бот не всегда это будет проверять, но иногда будет).



ДЗ №1: Simple Wall

Первая задача достаточно простая, у вас есть массив постов (он должен быть именно таким):

```
1  const posts = [  
2    {  
3      id: 3,  
4      type: 'text',  
5      content: 'Final Week!',  
6    },  
7    {  
8      id: 2,  
9      type: 'image',  
10     content: 'img/logo_js.svg',  
11   },  
12   {  
13     id: 3,  
14     type: 'video',  
15     content: 'video/video.mp4',  
16   },  
17 ];
```



ДЗ №1: Simple Wall

И две функции:

```
function makePostEl(post) {  
  | /* ваш код */  
}  
  
function makeWall(el, items) {  
  | items.map(makePostEl).forEach(/* ваш код */);  
}  
  
makeWall(rootEl, posts);
```

Функция `makePostEl` должна создавать из поста элемент и возвращать его.

Элемент создаётся исходя из типа поста.



ДЗ №1: Simple Wall

Вот такая разметка должна генерироваться для разных типов:

```
<div data-type="text" data-id="3">
|   <div>Final Week!</div>
</div>
<div data-type="image" data-id="2">
|   
</div>
<div data-type="video" data-id="1">
|   <video src="video/video.mp4" controls></video>
</div>
```



ДЗ №1: Simple Wall

Функция `makeWall` должна массив постов преобразовать в массив элементов, а затем все эти элементы разместить внутри `rootEl`:

```
<div id="root">
  <div data-type="text" data-id="3">
    <div>Final Week!</div>
  </div>
  <div data-type="image" data-id="2">
    
  </div>
  <div data-type="video" data-id="1">
    <video src="video/video.mp4" controls></video>
  </div>
</div>
```

Использование `map` и `forEach` обязательно – бот за этим будет следить.

Важно: бот будет запускать ваши функции со своими данными и проверять, работают ли они. Поэтому не завязывайтесь только на свои данные.



ДЗ №1: Simple Wall

В архиве должен быть каталог [simple-wall](#) с вашим проектом.



ДЗ №2: Comments

Что нужно сделать: нужно создать формочку для комментариев, с возможностью их динамического добавления в список:

A form consisting of a rectangular text input field on the left and a button labeled "Добавить" (Add) on the right. The button has a light gray background and a thin border.

- Первый комментарий
- Второй комментарий



ДЗ №2: Comments

Для этого ваше приложение должно генерировать следующую разметку:

```
<div id="root">
  <form data-id="comment-form">
    <textarea data-input="comment"></textarea>
    <button data-action="add">Добавить</button>
  </form>
  <ul data-id="comment-list">
    <li data-comment-id="1">Первый комментарий</li>
    <li data-comment-id="2">Второй комментарий комментарий</li>
  </ul>
</div>
```

Обратите внимание, комментариям присваиваются идентификаторы по порядку в виде целых чисел. Самый новый комментарий в самом низу.



ДЗ №2: Comments

Сами комментарии должны храниться в массиве `comments`. У комментария должны быть только свойства `id` (число) и `text` (строка).

Идентификатор можно генерировать с помощью оператора `++`:

```
let nextId = 1;

const comments = [
  {
    id: nextId++,
    text: 'Первый комментарий',
  },
];
```

При загрузке массив должен быть пустой. Каждое нажатие на кнопку "Добавить" должно приводить к добавлению в массив и к созданию элемента в DOM дереве.



ДЗ №2: Comments

В архиве должен быть каталог `comments` с вашим проектом.



ДЗ №3: Cashback

Что нужно сделать: сделайте форму, в которую пользователь может вносить название покупки и её стоимость. При каждом добавлении автоматически (через `reduce`) рассчитывается стоимость кэшбека. Будем считать, что гарантированный кэшбек составляет 0.5%.

<input type="text"/>	<input type="text"/>	<input type="button" value="Добавить"/>
----------------------	----------------------	---

- Вторая покупка на сумму 1000 с. (кэшбек - 5 с.)
- Первая покупка на сумму 2000 с. (кэшбек - 10 с.)

Итоговый кэшбек: 15 с.



ДЗ №3: Cashback

Ваше приложение должно генерировать следующую разметку:

```
<form data-id="purchase-form">
  <input data-input="name">
  <input data-input="price" type="number">
  <button data-action="add">Добавить</button>
</form>
<ul data-id="purchases-list">
  <li data-purchase-id="2">Вторая покупка на сумму 1000 с. (кэшбек - 5 с.)</li>
  <li data-purchase-id="1">Первая покупка на сумму 2000 с. (кэшбек - 10 с.)</li>
</ul>
<div>Итоговый кэшбек: <span data-id="total-cashback">15 с.</span></div>
```



ДЗ №3: Cashback

Все покупки должны содержаться в массиве `purchases` и иметь ровно три свойства:

1. `id` – число
2. `name` – строка (название, например, "Первая покупка")
3. `price` – число, (сумма, например, 1000)

В архиве должен быть каталог `cashback-calculator` с вашим проектом.



ДЗ №4: Purchases

Продолжим тему с покупками и **reduce**. Мы больше не хотим считать кэшбек, мы хотим вычислять самую дорогую покупку. Что это значит? Это значит, что когда вы заполняете форму, при каждом добавлении высчитывается самая дорогая:

<input type="text"/>	<input type="text"/>	<input type="button" value="Добавить"/>
----------------------	----------------------	---

- Вторая покупка на сумму 1000 с.
- Первая покупка на сумму 2000 с.

Самая дорогая покупка: Первая покупка на сумму 2000 с.



ДЗ №4: Purchases

Важно, если будет несколько покупок, то ваша программа в качестве самой дорогой должна считать ту, которая была добавлена последней (т.е. с наибольшим **id**):

<input type="text"/>	<input type="text"/>	Добавить
----------------------	----------------------	----------

- Третья покупка на сумму 2000 с.
- Вторая покупка на сумму 1000 с.
- Первая покупка на сумму 2000 с.

Самая дорогая покупка: Третья покупка на сумму 2000 с.



ДЗ №4: Purchases

Кнопка "Удалить" должна удалять покупку и из массива и из DOM-дерева. При этом самая дорогая покупка должна пересчитываться:

<input type="text"/>	<input type="text"/>	Добавить
----------------------	----------------------	----------

- Вторая покупка на сумму 1000 с.
- Первая покупка на сумму 2000 с.

Самая дорогая покупка: Первая покупка на сумму 2000 с.



ДЗ №4: Purchases

При первой загрузке покупок быть не должно и вместо названия покупки должно выводиться:

<input type="text"/>	<input type="text"/>	<input type="button" value="Добавить"/>
----------------------	----------------------	---

Самая дорогая покупка: нет покупок

То же самое сообщение должно выводиться, если из списка удалить все покупки.



ДЗ №4: Purchases

Все покупки должны содержаться в массиве `purchases` и иметь ровно три свойства:

1. `id` – число
2. `name` – строка (название, например, "Первая покупка")
3. `price` – число, (сумма, например, 1000)



ДЗ №4: Purchases

Бота устроит, что ваше приложение динамически генерирует следующую разметку:

```
<div id="root">
  <form data-id="purchase-form">
    <input data-input="name">
    <input data-input="price" type="number">
    <button data-action="add">Добавить</button>
  </form>
  <ul data-id="purchases-list">
    <li data-purchase-id="3">Третья покупка на сумму 2000 с. <button data-action="remove">Удалить</button></li>
    <li data-purchase-id="2">Вторая покупка на сумму 1000 с. <button data-action="remove">Удалить</button></li>
    <li data-purchase-id="1">Первая покупка на сумму 2000 с. <button data-action="remove">Удалить</button></li>
  </ul>
  <div>Самая дорогая покупка: <span data-id="most-expensive">Третья покупка на сумму 2000 с.</span></div>
</div>
```


В архиве должен быть каталог `purchases` с вашим проектом.




ДЗ №5: DownVoter

В большинстве социальных сетей есть функции голосования или аналогичные им (например, лайки/дизлайки). И достаточно часто негативная оценка пользователей служит тому, что "заминусованный" комментарий просто не показывается. Мы сделаем чуть проще и скажем, что заминусованный комментарий вообще должен удалиться.

Добавить

- Первый комментарий  0

+

-
- Второй комментарий  0

+

-

По умолчанию, когда комментарий добавляется, у него 0 лайков. Если количество лайков достигнет -10, то комментарий должен удалиться, как из массива, так и из DOM.



ДЗ №5: DownVoter

Сами комментарии должны храниться в массиве `comments`. У комментария должны быть только свойства `id` (число), `text` (строка) и `likes` (число).

Бота устроит следующая разметка (мы специально вынесли сердечко – вы можете брать любое):

```
<div id="root">
  <form action="" data-id="comment-form">
    <textarea name="" id="" cols="30" rows="10" data-input="text"></textarea>
    <button data-action="add">Добавить</button>
  </form>
  <ul data-id="comments-list">
    <li data-comment-id="1">
      <span data-info="text">Первый комментарий</span> ♥<span data-info="likes">0</span>
      <button data-action="like">+</button>
      <button data-action="dislike">-</button>
    </li>
    <li data-comment-id="2">
      <span data-info="text">Второй комментарий</span> ♥<span data-info="likes">0</span>
      <button data-action="like">+</button>
      <button data-action="dislike">-</button>
    </li>
  </ul>
</div>
```



ДЗ №5: DownVoter

В архиве должен быть каталог [downvoter](#) с вашим проектом.



ДЗ №6: Priority List

Ни одно обучение JS не обходится без TODO листа (списка дел). Но нам не интересно делать обычный, мы с вами сделаем TODO лист с приоритетами.

Как это выглядит: у вас есть форма, в которую вводится название и приоритет:

Название	<input type="text"/>
Приоритет	<input type="text"/>
<input type="button" value="Добавить"/>	



ДЗ №6: Priority List

Когда вы добавляете элементы, они выстраиваются по очереди в порядке приоритета (1-ый – самый важный приоритет):

Название

Приоритет

- Выучить JS (приоритет: 1)
- Выучить CSS (приоритет: 2)
- Выучить HTML (приоритет: 3)



ДЗ №6: Priority List

Но вы с помощью кнопок **+** и **-** можете увеличивать и уменьшать приоритет соответственно (приоритет не может быть меньше 1). При "увеличении приоритета" элементы должны перестраиваться:

Название

Приоритет

Название

Приоритет

- Выучить JS (приоритет: 1)
- Выучить HTML (приоритет: 1)
- Выучить CSS (приоритет: 2)

- Выучить JS (приоритет: 1)
- Выучить CSS (приоритет: 2)
- Выучить HTML (приоритет: 3)



Расположение элементов с одним приоритетом относительно друг друга – не принципиально.

Используйте метод **insertBefore** для "перестановки" элементов.



ДЗ №6: Priority List

Бота устроит
следующая разметка:

```
<div id="root">
  <form data-id="todo-form">
    <div>
      <label for="todo-text">Название</label>
      <input data-input="text" id="todo-text">
    </div>
    <div>
      <label for="todo-priority">Приоритет</label>
      <input data-input="priority" id="todo-priority" type="number">
    </div>
    <button data-action="add">Добавить</button>
  </form>
  <ul data-id="todo-list">
    <li data-todo-id="1">
      Выучить JS (приоритет: <span data-info="priority">1</span>)
      <button data-action="inc">+</button>
      <button data-action="dec">-</button>
    </li>
    <li data-todo-id="3">
      Выучить HTML (приоритет: <span data-info="priority">1</span>)
      <button data-action="inc">+</button>
      <button data-action="dec">-</button>
    </li>
    <li data-todo-id="2">
      Выучить CSS (приоритет: <span data-info="priority">2</span>)
      <button data-action="inc">+</button>
      <button data-action="dec">-</button>
    </li>
  </ul>
</div>
```

ДЗ №6: Priority List

Сами задачи должны храниться в массиве `tasks`. У задачи должны быть только свойства `id` (число), `text` (строка) и `priority` (число).

Подсказка: обратите внимание, что задача устроена хитро – кнопка `dec` с одной стороны "уменьшает" приоритет как число, но чем меньше число, тем, на самом деле, выше приоритет.

Подсказка по реализации: вы можете по массиву задач определить, в какую позицию вставлять новую задачу, а потом в DOM вставлять на эту же позицию. То же самое касается изменения позиции.

В архиве должен быть каталог `priority-list` с вашим проектом.



ДЗ №7: Шаблонизатор

Мы хотим с вами написать функцию-шаблонизатор. Что это такое? Это функция, которая принимает на вход объект, оформленный в определённом виде и возвращает `HTMLElement`.

Функция должна называться `makeElement`, принимать один параметр `el`, и работать следующим образом: на вход функции поступают объекты вида:

```
const obj = {  
  tagname: 'div',  
  attributes: {  
    id: 'first',  
    'data-id': 'first',  
    'class': 'primary',  
  },  
  text: 'Hello, JS',  
};
```



ДЗ №7: Шаблонизатор

Что есть что:

1. **tagname** – обязательно, какой элемент нужно создавать
2. **attributes** – какие атрибуты нужно назначать
3. **text** – какое текстовое содержимое должно быть внутри (**textContent**) – этого свойства может не быть

```
const obj = {  
  tagname: 'div',  
  attributes: {  
    id: 'first',  
    'data-id': 'first',  
    'class': 'primary',  
  },  
  text: 'Hello, JS',  
};
```



ДЗ №7: Шаблонизатор

Обратите внимание, что некоторые свойства обрамлены в кавычки – это потому, что без кавычек они являются невалидными именами свойств для выставления атрибутов (т.е. если их просто прописать в таком же виде в элементы, мы не получим желаемого результата).

```
const obj = {  
  tagname: 'div',  
  attributes: {  
    id: 'first',  
    'data-id': 'first',  
    'class': 'primary',  
  },  
  text: 'Hello, JS',  
};
```



ДЗ №7: Шаблонизатор

Вам нужно будет прочитать про объект, который называется **Object** и найти в нём методы, которые позволяют извлекать из объектов имена свойств и их значения.

Обратите внимание, бот обязательно будет смотреть, как вы будете обрабатывать **data**-атрибуты и атрибут с именем **class** (ради справедливости нужно отметить, что в объекте, не элементе, а именно объекте, свойство **class** можно передавать без кавычек).

В архиве должен быть каталог **make-element** с вашим проектом.



Приведение типов



Приведение типов

1. Числовое
2. Строковое
3. Boolean



Числовое

Выполняется при:

- `Number(<expr>)`
- арифметических операторах



Числовое

- undefined -> NaN
- null -> 0
- true -> 1
- false -> 0
- " ... " -> в число или в NaN:
 - начальные и конечные пробелы не учитываются
 - парсится число (0 - если пусто, число - если только число, NaN - если не удалось обработать)



Числовое

- " 23 " -> 23
- " 023 " -> 23
- " 23X" -> NaN
- "" -> 0

Иногда для преобразования в число используют унарный плюс: +"23"



Строковое

- `String(<expr>)`
- `+` с операндом-строкой (бинарный)
- все операции, требующие строку



Строковое

- undefined -> 'undefined'
- null -> 'null'
- false -> 'false'
- true -> 'true'
- и т.д.



Boolean

- `Boolean(<expr>)`
- Синтаксические конструкции (if, for, while)

Иногда используют `!!<expr>`, чтобы быстро привести значение к `boolean`

`!` – оператор отрицания, из `true` делает `false`, из `false` делает `true`. `!!` – двойное отрицание (сначала приводим значение к противоположному, но типа `boolean`, а затем обратно).



Falsy values

В false преобразуются следующие значения:

- `undefined`
- `null`
- `0`
- `NaN`
- `0n`
- пустая строка (строка, в которой нет ни одного символа)



Спасибо за внимание

alif skills

2023г.

